Accelerating a 3D finite-difference wave propagation code using GPU graphics cards

David Michéa, Dimitri Komatitsch

Geophys. J. Int. (0000) **000**, 000–000

Accelerating a 3D finite-difference wave propagation code using GPU graphics cards

David Michéa[1] and Dimitri Komatitsch[2,3]

[1] *Bureau de Recherches Géologiques et Minières, 3 avenue Claude Guillemin, BP 36009, 45060 Orléans Cedex 2, France, E-mail: d.michea@brgm.fr*
[2] *Université de Pau et des Pays de l'Adour, CNRS & INRIA Magique-3D, Laboratoire de Modélisation et d'Imagerie en Géosciences UMR 5212, Avenue de l'Université, 64013 Pau Cedex, France. E-mail: dimitri.komatitsch@univ-pau.fr*
[3] *Institut universitaire de France, 103 boulevard Saint-Michel, 75005 Paris, France.*

30 November 2010

SUMMARY
We accelerate a three-dimensional finite-difference in the time domain (FDTD) wave propagation code by a factor between about 20 and 60 compared to a serial implementation using Graphics Processing Unit (GPU) computing on NVIDIA graphics cards with the CUDA programming language. We describe the implementation of the code in CUDA to simulate the propagation of seismic waves in a heterogeneous elastic medium. We also implement Convolution Perfectly Matched Layers (CPMLs) on the graphics cards to efficiently absorb outgoing waves on the fictitious edges of the grid. We show that the code that runs on a graphics card gives the expected results by comparing our results to those obtained by running the same simulation on a classical processor core. The methodology that we present can be used for Maxwell's equations as well because their form is similar to that of the seismic wave equation written in velocity vector and stress tensor.

1 INTRODUCTION

Finite-difference (FD) techniques in the time domain (FDTD) are widely used to solve wave equations such as Maxwell's equations (Yee 1966) or the seismic wave equation (Alterman & Karal 1968; Madariaga 1976; Virieux 1986), and have been used to solve Navier-Stokes' equations as well (Chorin 1968). For a recent thorough review on FD applied to the seismic wave equation see e.g. Moczo et al. (2007). When more geometrical flexibility is needed for instance to handle geometrically complex models other techniques such as a pseudospectral technique (e.g., Carcione & Wang (1993); Komatitsch et al. (1996)), a boundary-element method (e.g., Kawase (1988); Vai et al. (1999)), a spectral-element method (e.g., Liu et al. (2004); Chaljub et al. (2007); Tromp et al. (2008)) or a discontinuous Galerkin technique (e.g., Reed & Hill (1973); Falk & Richter (1999); Hu et al. (1999); Cockburn et al. (2000); Giraldo et al. (2002); Rivière & Wheeler (2003); Monk & Richter (2005); Grote et al. (2006); Bernacki et al. (2006); Dumbser & Käser (2006)) are sometimes needed, but because of its simplicity FD remains very widely used.

FD is often used in conjunction with a Perfectly Matched Layer (PML) to absorb waves on the artificial edges of the numerical grid to mimic an infinite or semi-infinite medium (see e.g. Bérenger (1994) for Maxwell's equations and Hastings et al. (1996); Chew & Liu (1996); Collino & Tsogka (2001) for the seismic wave equation); or even better a Convolution PML (CPML) can be used to improve the behavior of the discrete PML for waves impinging the artificial edges of the grid at grazing incidence (see e.g. Roden & Gedney (2000) for CPML for Maxwell's equations, Festa & Vilotte (2005); Festa et al. (2005) for split CPMLs for seismic waves and Martin et al. (2005); Martin & Komatitsch (2006); Komatitsch & Martin (2007); Drossaert & Giannopoulos (2007); Martin et al. (2008a,b); Martin & Komatitsch (2009); Kristek et al. (2009) for unsplit CPMLs for seismic waves. More recently a formulation of the unsplit CPML that can easily be extended to higher-order time schemes, called the Auxiliary Differential Equation PML (ADE-PML), has been introduced by Gedney & Zhao (2010) for Maxwell's equations and by Martin et al. (2010) for the seismic wave equation. An improved sponge layer, improperly called the split Multiaxial PML (M-PML), has been suggested by Meza-Fajardo & Papageorgiou (2008), but the perfectly matched character of Bérenger (1994) is lost because of the coupling introduced between derivatives along more than one grid axis.

In recent years, computing on graphics cards (also known as 'Graphics Processing Unit (GPU) computing') has been used to accelerate non-graphical applications with respect to calculations performed on a classical Central Processing Unit (CPU) processor core (see e.g. Owens et al. (2007) or Kirk & Hwu (2010)). A number of physical problems have been solved, e.g., molecular dynamics simulations (Yang

et al. 2007; Anderson et al. 2008), fluid dynamics simulations (Brandvik & Pullan 2007; Elsen et al. 2008), or astrophysical calculations (Nyland et al. 2007). Regarding FD, several applications have been ported to GPUs as early as 2004 (Krakiwsky et al. 2004a,b; Baron et al. 2005; Humphrey et al. 2006; Adams et al. 2007; Price et al. 2007; Abdelkhalek 2007; Inman et al. 2007; Balevic et al. 2008a,b; Valcarce et al. 2008; Inman & Elsherbeni 2008; Micikevicius 2009; Abdelkhalek et al. 2009). Some of the other numerical techniques mentioned above for seismic wave propagation have recently been successfully ported to GPUs, for instance the spectral-element method by Komatitsch et al. (2009) in the case of one GPU and by Komatitsch et al. (2010a,b) in the case of a cluster of many GPUs used in parallel, and the discontinuous Galerkin method by Klöckner et al. (2009).

GPU programming on NVIDIA graphics cards has become significantly easier with the introduction at the end of 2006 of the CUDA programming language (NVIDIA Corporation 2009a), which is relatively easy to learn because its syntax is similar to C. Regarding FD for seismic reverse time migration in the case of an acoustic medium with constant density, Abdelkhalek (2007) and Micikevicius (2009) (from NVIDIA corporation, the developers of GPU hardware and of CUDA) have recently introduced optimized implementations. Abdelkhalek et al. (2009) extended it to the acoustic case with heterogeneous density. In this article, we use CUDA to solve the seismic wave equation in the more complex fully heterogeneous (including for density) elastic case on several GPUs used in parallel. We improve upon Micikevicius (2009) by applying his methodology to the full elastic wave equation and to a real three-dimensional (3D) model, by adding CPML absorbing layers, by making access to these CPML arrays faster on the GPU by resorting to so-called 'CUDA texture fetching' to access them and overcome the limited amount of shared memory that is available on the GPU hardware, and finally by using message passing (MPI, see e.g. Gropp et al. (1994)) to use several GPUs in parallel efficiently.

2 THE SEISMIC WAVE EQUATION AND A CLASSICAL FINITE-DIFFERENCE DISCRETIZATION

We consider a linear isotropic elastic rheology for the solid medium, and therefore the seismic wave equation can be written in the strong, i.e., differential, form

$$\begin{aligned} \rho \ddot{\mathbf{u}} &= \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \mathbf{f} \,, \\ \boldsymbol{\sigma} &= \mathbf{c} : \boldsymbol{\varepsilon} \,, \\ \boldsymbol{\varepsilon} &= \tfrac{1}{2} [\boldsymbol{\nabla} \mathbf{u} + (\boldsymbol{\nabla} \mathbf{u})^T] \,, \end{aligned} \tag{1}$$

where $\mathbf{u}$ denotes the displacement vector, $\boldsymbol{\sigma}$ the symmetric, second-order stress tensor, $\boldsymbol{\varepsilon}$ the symmetric, second-order strain tensor, $\mathbf{c}$ the fourth-order stiffness tensor, ρ the density, and $\mathbf{f}$ an external source force. The double tensor contraction operation is denoted by a colon, a superscript T denotes the transpose, and a dot over a symbol indicates time differentiation. The physical domain of the model is denoted by Ω and its outer boundary by Γ. The material properties of the solid, $\mathbf{C}$ and ρ, can be spatially heterogeneous.

In the classical velocity-stress formulation that is used in most FD implementations (e.g., Collino & Tsogka 2001; Moczo et al. 2007), one rewrites Eq. (1) as a first-order system whose unknowns are the velocity vector $\mathbf{v}$ and the stress tensor $\boldsymbol{\sigma}$:

$$\begin{aligned} \rho \partial_t \mathbf{v} &= \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}, \\ \partial_t \boldsymbol{\sigma} &= \mathbf{c} : \boldsymbol{\nabla} \mathbf{v} \,. \end{aligned} \tag{2}$$

The boundary condition at the free surface of the medium is that the traction vector $\boldsymbol{\tau}$ must be zero everywhere at the surface Γ, i.e.,

$$\boldsymbol{\tau} = \boldsymbol{\sigma} \cdot \hat{\mathbf{n}} = \mathbf{0} \,, \tag{3}$$

where $\hat{\mathbf{n}}$ is the outgoing normal to the surface Γ.

Discretization of the first-order system (2) together with the free surface boundary condition (3) is then classically performed based on the staggered grid of Figure 1, which was introduced for Maxwell's equations by Yee (1966) and for the seismic wave equation by Madariaga (1976). Our goal here is not to describe in detail the classical finite-difference discretization of the seismic wave equation written in velocity vector and stress tensor because such a description can be found in numerous textbooks and articles (see e.g. Collino & Tsogka (2001), and Moczo et al. (2007) for a thorough review). Let us just recall that in the finite-difference method partial spatial derivatives are approximated by discrete operators involving differences between adjacent grid points, as in the stencil of Figure 2, which represents the fourth-order spatial operator that we use in our 'ONDES3D' software package (Aochi & Douglas 2006). As an example, the spatial first derivative $\partial_x u$ of a given $u(x,y,z,t)$ component of a field along the x axis of a regular 3D grid is approximated based upon a Taylor expansion

$$\partial_x u(x + \frac{\Delta x}{2}, y, z, t) \simeq \frac{9}{8} . \frac{u(x + \Delta x, y, z, t) - u(x, y, z, t)}{\Delta x} - \frac{1}{24} . \frac{u(x + 2\Delta x, y, z, t) - u(x - \Delta x, y, z, t)}{\Delta x} \,, \tag{4}$$

where Δx is the size of an elementary grid cell along the x axis, which means that at a given time t the derivative can be computed numerically by using:

$$\partial_x u(i + \frac{1}{2}, j, k) \simeq \frac{9}{8} . \frac{u(i+1,j,k) - u(i,j,k)}{\Delta x} - \frac{1}{24} . \frac{u(i+2,j,k) - u(i-1,j,k)}{\Delta x} \,. \tag{5}$$

In practice people often resort to optimized spatial coefficients designed to minimize overall numerical dispersion, for instance those of Holberg (1987), but this has no influence on the GPU implementation discussed herein. At the free surface of the model, in the vertical

direction we switch to a second-order spatial operator in order to be able to implement the free surface condition, as done classically (see e.g. Moczo et al. (2007)). Time evolution is performed based on a staggered central finite-difference approximation, as in Virieux (1986).

In regional or local seismology in many cases one is interested in simulating a semi-infinite medium with a free upper surface. All the edges of the grid except the top edge are then artificial and outgoing waves should be absorbed there in order to simulate a semi-infinite medium. We use the unsplit CPML technique of Komatitsch & Martin (2007), also analyzed by Kristek et al. (2009), which consists in modifying each spatial derivative along the direction perpendicular to the absorbing layer, say x, in the following fashion:

$$\partial_{\tilde{x}} = \frac{1}{\kappa_x}\partial_x + \psi_x \ , \tag{6}$$

where ψ_x is a memory variable whose time evolution is governed at each time step by an additional equation:

$$\psi_x^n = b_x \psi_x^{n-1} + a_x \left(\partial_x\right)^{n-\frac{1}{2}} \ . \tag{7}$$

This implies that significantly more equations need to be solved in the PML regions, in particular near the corners of the 3D grid, because contributions coming from the PML layers located along x, y and z are summed there and one memory variable and thus a time evolution equation is needed for each; but this is acceptable because the PML regions are small compared to the rest of the model.

Coefficients a_x and b_x in the PML, which do not vary with time, are given by:

$$b_x = e^{-(d_x/\kappa_x + \alpha_x)\Delta t} \tag{8}$$

and

$$a_x = \frac{d_x}{\kappa_x(d_x + \kappa_x \alpha_x)}(b_x - 1) \ , \tag{9}$$

where $\kappa_x \geq 1$, $d_x \geq 0$ and $\alpha_x \geq 0$ are three real damping coefficients. We refer the reader to Komatitsch & Martin (2007) for more details. Note that if a higher-order time scheme were used, one should resort to an Auxiliary Differential Equation (ADE) implementation of the PML optimized at grazing incidence instead of a convolutional implementation, as introduced by Martin et al. (2010).

3 IMPLEMENTATION ON GRAPHICS CARDS USING CUDA

Let us first summarize a few key concepts regarding programming GPU graphics cards with CUDA, and then see how to port our elastic wave propagation code to it.

3.1 A brief summary of GPU programming concepts

For readers not familiar with details of CUDA or GPU programming, let us briefly explain the programming model that supports the fine-grained parallel architecture of NVIDIA GPUs. Considering the potentially very high performance increase that one may get for a wide range of applications by porting them to GPUs, it is of interest to become familiar with these new programming concepts, which are significantly different from classical serial programming in Fortran or C on a CPU. In the glossary of Table 1 we briefly explain some of the terms most commonly used in the context of graphics cards and CUDA and that are used several times in the rest of the article. For more details the reader is referred to the CUDA documentation (NVIDIA Corporation 2009a) and GPU/CUDA conference tutorials (see e.g. http://gpgpu.org/developer).

The official CUDA documentation (NVIDIA Corporation 2009a) and publications related to CUDA (see Section 1) often use varying terminologies, in particular when defining the notion of a computing 'core' on GPUs. In this article, we identify each so-called 'multiprocessor' of a GPU with a 'Single Instruction, Multiple Data (SIMD) core'. The individual thread processors within each multiprocessor, which are called 'streaming processor cores' or 'CUDA cores' in NVIDIA literature, share the instruction stream and can therefore be viewed as arithmetic logic units (ALUs).

The code that gets executed on the GPU is called a calculation 'kernel'. This kernel is launched on a grid of thread blocks. Threads inside the same block can be synchronized, but no synchronization is possible between blocks. The different blocks are placed on the different multiprocessors by a scheduler that removes stalled blocks waiting for input to or output from memory and launches thread blocks that are ready for execution. The same physical multiprocessor can execute several blocks, and the order in which blocks are assigned to multiprocessors is undefined. There is always an implicit synchronization between kernel calls on dependent data (i.e., when some of the output of one kernel is used as input to the next).

The memory available on the graphics card is distributed between threads and thread blocks. Each thread has its own registers, whose access is extremely fast but whose total number is limited. Each thread block can use a small amount of low-latency on-chip 'shared memory', which can be read from and written to by all the threads of the same block. This is an efficient way for threads within one block to exchange data. Read/write access to it is always very fast, however the cost of some access patterns is higher than others.

In numerical techniques such as the finite-difference method that compute values on a grid of mesh points, in a classical programming model on a CPU calculations are performed sequentially for each point by the same thread. The programming model on a GPU is very different: the calculations are going to be done in parallel by an extremely high number of very lightweight threads, typically hundreds of

thousands. Often in an FD code one decides to use one thread per mesh point. As seen above, in CUDA these threads are grouped in thread blocks, and these blocks are grouped in a grid of blocks. The threads within one block are assigned to consecutive so-called 'warps' of 32 threads.

Central memory (called 'global memory') is common to all the threads and is the place where one stores arrays and data that need to be shared between different kernels. In current GPU hardware it does not possess any cache mechanism and input/output to it is very slow ($\simeq$500-600 clock cycles). For read-only accesses, global memory can be read through a so-called 'texture fetching' mechanism that has a cache.

The threads of a given block are executed on the vectorial processors (the multiprocessors) of the graphics card by groups of 16, called a 'half-warp'. Accesses to global memory can be 'coalesced' automatically by the GPU hardware into, in the best case, a single, large and efficient memory transaction per half-warp of 16 threads. Whether this happens depends on a number of relatively severe restrictions on the memory access pattern and on data alignment (see e.g. Bell & Garland (2009)); in such a case, memory performance is maximized.

If the code that is run by the 16 threads of a half-warp is the same and if the data that they need can be accessed in a coalesced fashion, the half-warp is executed fully in parallel in one pass, leading to optimal performance. Otherwise the half-warp is called 'divergent' and is cut automatically by the system into non-diverging pieces that are executed one after the other, i.e., serialized, leading to reduced performance.

Each multiprocessor can handle 512 threads simultaneously, switching to the next available warp when the currently executed one stalls, for instance because it is waiting for input from or output to memory. These switches are extremely fast, with almost no overhead, because they are performed directly by the hardware of the graphics card and not by software. The actual number of thread blocks that run simultaneously is determined by the amount of resources used, in particular registers and shared memory. This is the way a GPU maximizes the utilization of its multiprocessors and thus the overall execution speed.

The key issue to get an efficient code on a GPU is to manage to overlap the huge access time (latency time) to data stored in memory by always having blocks ready to be computed while other blocks are waiting for input from or output to memory. This implies having a large-enough number of blocks to schedule, and also minimizing accesses to global memory, in particular redundant reads inside a block of threads, by loading these data once to shared memory, from which all the threads of a thread block will then be able to read them in a much faster fashion.

One of the main drawbacks of current GPUs is the fact that the amount of shared memory and the number of registers is drastically limited. This limits the number of threads and of thread blocks that can run simultaneously on the graphics card. The ratio between the number of thread blocks that run simultaneously in a given kernel and the theoretical maximum number of thread blocks that the hardware could in principle run simultaneously is called the 'multiprocessor occupancy' or simply the 'occupancy' in the CUDA literature. It can be viewed as the fraction of the theoretical peak computation power of the graphics card that is really used by the application. When porting an application to a GPU, it is therefore important to try to maximize the occupancy by minimizing the resources used by the threads and thread blocks.

However in practice it can be difficult to reach high occupancy in some parts of a complex application because often a significant amount of shared memory must be used, and it may be very difficult to reduce the total number of registers used because it is defined mostly by the CUDA compiler itself without any control on it from the user. This issue is not specific to GPU computing: on classical processors there is often a large difference between the theoretical peak performance and the sustained performance of the running application, i.e., only a small part of the theoretical peak computational power is actually used. Let us also note that in many applications the performance of the code is limited by the bandwidth of accesses to memory and not by the speed of the calculations, for instance in applications in which only a small or medium number of calculations is performed on each value read from memory, which is a common case. Such codes are called 'bandwidth bound'. In such a case, trying to increase occupancy in a CUDA kernel might be counterproductive because if more blocks can be executed simultaneously then there is more latency in memory accesses that must be overlapped, which is difficult. Thus, and contrary to what people do when programming classical CPUs, one can often increase performance on a GPU by purposely recomputing things that are needed several times in an algorithm rather than storing them in memory, in order to increase the calculations/memory accesses ratio.

3.2 Porting our elastic wave propagation algorithm to a GPU

The main difficulty when implementing a finite-difference code on a GPU comes from the stencil of Figure 2. For a fourth-order spatial operator, the thread that handles the calculation of point (i, j, k) needs to know the fields (and therefore access the arrays) at point (i, j, k) but also $(i + 1, j, k)$, $(i + 2, j, k)$, $(i - 1, j, k)$, $(i - 2, j, k)$, $(i, j + 1, k)$, and so on, thus of the point that it handles and of 12 neighboring points. This implies that 13 accesses to global memory are needed on the GPU to handle each grid point, which is a very high value, keeping in mind that access to global memory is very slow as seen above.

But because threads that belong to the same block of threads can access common values using much faster shared memory, it is possible to significantly reduce this number of memory accesses per grid point and thus drastically improve performance. Since we use an explicit time scheme, values computed in the whole grid at a given time t depend only on past values already computed and are therefore independent. We can thus implement spatial parallelism by computing many grid points in parallel. We have seen above that because GPUs require massive multithreading based on very lightweight threads, we will use a different thread to handle each grid point. We have also seen that, in CUDA, threads are grouped in thread blocks. The most intuitive approach is to use a cubic distribution of threads (Figure 3(a)). This way, each thread will load the values of the arrays at the grid point it handles from global memory to shared memory, and thus inside a given block of threads

many neighboring points of the finite-difference stencil of Figure 2 are automatically loaded to shared memory by the other threads of the block because with such a cubic distribution of threads by definition they handle these neighboring points. However, for points located on the edges of the block, some of the neighboring points do not belong to the block but rather to an external 'halo' drawn in Figure 3(a). These halo points also need to be loaded to shared memory, which results in a small number of additional reads from global memory for some of the threads of the thread block. The intuitive approach of using a cubic distribution of threads allows one to reduce the ratio between the number of grid points and the size of the halo, as illustrated in Figure 3(a). Unfortunately in practice on current GPUs there is not enough shared memory to allow one to use 3D blocks large enough to sufficiently reduce this ratio, and therefore this approach cannot be efficiently implemented.

We therefore turn to a more efficient 2D approach, introduced by Micikevicius (2009), which uses a sliding computation window. This idea is somewhat similar to the approach of Graves (1996) to run large 3D FD simulations on a computer that does not have enough memory by storing parts of the grid to external storage, for instance a hard drive, a technique known in computer science as 'out of core' storage.

Instead of subdividing the 3D volume into 3D sub-volumes with halos and distributing them along the grid of thread blocks for a one pass calculation, as done in the 3D approach, the 2D approach consists in tiling a 2D cut plane of the volume (for instance in the X and Y directions of the finite-difference grid) with 2D tiles, each tile corresponding to a block of threads. One can then iterate along the third (and last) direction, e.g. the Z direction, shifting the halo points for this last direction in registers organized in a pipeline fashion, taking advantage of the fact that access to these registers is extremely fast. The data of the 2D mesh tile and its halos (Figure 3(b)) are loaded in shared memory from global memory. To do this, each thread first loads the data corresponding to the grid point it handles into shared memory, and then the halos are loaded in a second step. We use four registers per thread to store the data values corresponding to points located in $k+1$, $k+2$, $k-1$ and $k-2$. We then iterate on k and use these four registers as a shifting register (or equivalently a one-dimensional sliding window or pipeline), loading the new value in $k+2$ and discarding the old value in $k-2$ at each iteration. It turns out that this approach is more efficient than the intuitive 3D approach. As we have seen, without any optimization 13 memory accesses are needed per grid point for each array that needs to be read in our FD scheme. With the intuitive 3D approach and for a spatial FD operator of order k and with cubic 3D threads blocks of size $n \times n \times n$ in Figure 3(a), $\frac{(n^3+3kn^2)}{n^3}$ memory accesses per grid point and per array read are needed. If we compare the number of accesses needed per grid point and per array read between the 3D approach with cubic blocks of 125 threads, i.e., $n = 5$, and the 2D approach with blocks of $16 \times 8 = 128$ threads, for our fourth-order operator ($k = 4$) and for $n = 5$ we get 3.4 memory accesses per grid point, and for our implementation of the 2D approach we need only $((8 \times 16) + k \times 8 + k \times 16)/(8 \times 16) = 1.75$ memory accesses per grid point, i.e., approximately twice fewer. In spite of this drastic reduction, a value of 1.75 still implies that the code will be bandwidth bound because the number of (slow) memory accesses per grid point will not be very small compared to the number of (very fast) calculations performed at that point, i.e., comparatively slow memory accesses will be the main limiting factor. This is intrinsically related to the finite-difference method and nothing more can be done to avoid it. Let us note that NVIDIA has announced their new 'FERMI' architecture (NVIDIA Corporation 2009b), which will have a cache system for access to global memory and which should therefore make this issue less critical.

The optimal size to use for the 2D blocks of threads depends strongly on the resources available on the GPU, in particular the amount of shared memory and the number of registers available. As we have seen above, using more registers and/or more shared memory per thread reduces the occupancy of the code on the GPU and may thus have a significant negative impact on performance. In our case, for a NVIDIA GeForce 8800 GTX video card, using the "CUDA GPU occupancy calculator" provided by NVIDIA in the CUDA software distribution we have been able to determine that blocks of $8 \times 16 = 128$ threads give the best result and an occupancy of $1/6 \simeq 0.167$, which is not too problematic, as seen above, because the code is bandwidth-bound.

Even if the above approach enables us to reduce the number of accesses to global memory, there are constraints imposed by the GPU hardware to get a good level of parallelism, in particular regarding the coalescence of accesses to global memory. On the NVIDIA 8800 GTX video card that we use, to get perfectly coalesced reads when accessing single-precision floats, first the 16 threads of a half-warp must access 32-bit data (one float), resulting in a 64-byte memory transaction (16 floats). Second, these 16 floats must reside in the same memory segment of 64 bytes, which implies that the address of the first float must be a multiple of 64 bytes, i.e., this is a memory alignment constraint. And third, the threads must access the floats in an ordered sequence: the k-th thread of the half-warp must access the k-th float. To honor these three constraints in the code, we align the first point of the mesh on a multiple of 64 bytes, using zero padding in memory if needed. This way, each thread automatically uses perfectly coalesced reads from and writes to the (i, j, k) grid point that it handles. This is true for all the points handled by the thread block but not for the points that belong to the halo (Figure 3(b)) because the reads for these halo points cannot honor the three above constraints.

To efficiently access the one-dimensional damping profile parameters in the CPML absorbing layers, which are read-only values that can be stored in one-dimensional arrays whose size is the thickness (in grid points) of each CPML layer, we use a so-called 'texture fetching' mechanism of CUDA, also called a one-dimensional texture, because it possesses a cache and therefore allows faster access to these arrays.

The code is implemented in single precision, which is sufficient in the vast majority of seismic modeling applications in the time domain based on an explicit time scheme (see e.g. Komatitsch & Tromp (1999) and Komatitsch et al. (2009) and benchmarks therein). The first advantage is reduced memory storage, by a factor of exactly two. Another advantage is faster calculations on the GPU, because according to the specifications given by the vendors, theoretical peak performance is between eight (on NVIDIA cards equipped with a GT200 chip, such as our 8800 GTX), five (AMD RV870 chip) and two (NVIDIA GT300 chip) times slower in double precision than in single precision.

However it is worth mentioning that in algorithms such as finite differences that are intrinsically memory bound this issue is less critical than it may sound because the main limitation is slow memory accesses, not the speed of calculations, as seen above.

Host to device (i.e., CPU to GPU) or device to host (i.e., GPU to CPU) data transfers often have a severe impact on performance in codes ported to a GPU because these transfers go through a PCI-Express bus on the motherboard, whose bandwidth is limited compared to the speed of execution of the code on the GPU. Fortunately this is not a problem in our FD code when a single GPU is used because the two main calculations in the algorithm, namely the calculation of the stress tensor and then the calculation of the velocity vector, which require more than 95% of the total number of operations, are performed entirely on the GPU without any transfer to/from the host. At each time step of the time loop, only one small transfer is required at the end in order to store that time step of the seismogram (i.e., the final result we are interested in) on the CPU, for instance to save it on the hard drive at the end of the simulation. Tests not shown here have demonstrated that this represents at most a few percent of the total execution time of the code and is therefore not a problem in practice.

3.3 Using several GPUs in parallel based on message passing (MPI)

Let us now explain how several GPUs can be used in parallel to perform the calculations. Micikevicius (2009) already presented a methodology to use up to 4 GPUs on a single node in the acoustic case with constant density, achieving linear scaling by overlapping inter-GPU communication with computation, and Abdelkhalek et al. (2009) described a way of running on a GPU cluster using overlapped non-blocking message passing communications between GPU nodes based on the Message-Passing Interface (MPI, see e.g. Gropp et al. (1994)). Here we extend such a message-passing approach to the heterogeneous elastic case with CPML absorbing layers that we have presented above for an implementation on a single GPU.

To implement MPI communications in our GPU application, we choose the simplest way: each GPU is managed by a separate CPU core. We do not use a more complex implementation that could for instance be based on threads, in which a CPU core could handle more than one GPU. We use non-blocking MPI communications between compute nodes and we implement a classical technique (see e.g. Danielson & Namburu (1998)) to overlap the communication time with computations performed on each GPU: we compute values for points that need to be exchanged with other GPUs first, i.e., points that are located along cut planes between adjacent mesh slices, and then we compute other (inner) points not shared with any neighbor while the non-blocking messages are traveling across the network. Achieving effective overlap requires that the ratio of the number of inner to outer grid points be sufficiently large, which is the case for large enough mesh slices. Under these conditions, the non-blocking MPI data transfer will statistically likely complete before the completion of the computation of the inner elements, and thus have negligible cost.

This classical approach can be decomposed into the following steps:

1) first compute values for points that need to be exchanged (i.e., the outer points of each MPI slice)

2) extract MPI buffers from the main arrays

3) launch asynchronous MPI ‘send’ and ‘receive’ calls for these buffers

4) compute values for the inner points inside each mesh slice while the non-blocking messages are traveling across the network

5) wait for all the communications to arrive (if effective overlap is achieved, this step has negligible cost)

6) update the main arrays based on the content of the MPI buffers received

In our multi-GPU code, we implement this algorithm with two more steps, as illustrated in Figure 4:

2bis) copy the MPI buffers created from a given GPU to the CPU core that handles it

5bis) copy the MPI buffers received from a given CPU core to the GPU it handles.

Because the arrays are stored on the GPU video card, steps 2) and 6) are performed on the GPUs using dedicated CUDA kernels.

4 NUMERICAL VALIDATION AND PERFORMANCE ANALYSIS

GPU hardware is available in two different flavors: relatively cheap video cards that can be installed on classical PCs that users already possess, and higher-end and more expensive dedicated graphics computing servers, for instance NVIDIA's TESLA machines. Let us study both categories for our finite-difference application, the first category targeting low cost and the second targeting more demanding applications, for instance calculations that require more memory. Our first experimental setup is therefore a cheap (its typical cost currently being around US$250) NVIDIA GeForce 8800 GTX video card installed in the PCI Express 1 bus of a quad-processor dual-core 64-bit AMD Opteron 2.8 GHz PC with 8 gigabytes of RAM memory and running Linux kernel 2.6.9. The 8800 GTX card has 16 multiprocessors, i.e., 128 cores, and 768 megabytes of memory. The bandwidth for the memory transfers is 86.4 gigabytes per second, with a memory bus width of 384 bits. Our second experimental setup is a cluster of 48 Teslas S1070 at CCRT/CEA/GENCI in Bruyères-le-Châtel, France; each Tesla S1070 has four GT200 GPUs and two PCI Express-2 buses (i.e., two GPUs share a PCI Express-2 bus). The GT200 cards have 4 GB of memory, and the memory bandwidth is 102 gigabytes per second with a memory bus width of 512 bits. The Teslas are connected to BULL Novascale R422 E1 nodes with two quad-core Intel Xeon Nehalem processors operating at 2.93 GHz. Each node has 24 GB of RAM and runs Linux kernel 2.6.18. The network is Infiniband.

First, on a single GT200 card, we determined experimentally (by trying to allocate increasingly larger chunks of memory) that out of the 4 gigabytes of memory that are installed on that GPU, approximately 3.9 gigabytes are available to the user (the rest is likely used by the

system to store the code itself, and also maybe for CUDA texture binding). The largest 3D model that we can thus use has a size of 53.9 km $\times$ 48.3 km $\times$ 18.8 km discretized using a grid of 540 $\times$ 484 $\times$ 189 points, i.e., with grid cells of size 100 m in the three spatial directions. The model is heterogeneous and composed of two horizontal elastic layers in contact. The lower layer has a pressure wave velocity of 6000 m.s^{-1}, a shear wave velocity of 3460 m.s^{-1}, and a density of 2700 kg.m^{-3}. The upper layer has a pressure wave velocity of 4200 m.s^{-1}, a shear wave velocity of 2422 m.s^{-1}, and a density of 2300 kg.m^{-3}. The interface between the two layers is located at a depth of 1400 m. The values of the pressure wave velocity, shear wave velocity and density are stored at each grid point; thus, more complex heterogeneous models could be handled in the context of real seismological applications. We take a time step of 8 ms and we simulate a total of 1000 time steps, i.e., a total duration of 8 s. The source is a moment tensor of strike 0°, dip 90° and rake 0° and scalar moment $M_0 = 10^{18}$ N.m located at x = 2000 m, y = 4000 m and z = -2500 m. The moment rate time function $m(t)$ is a Gaussian with spread τ:

$$m(t) = \frac{M_0 H(t)}{\sqrt{2\pi}\tau} e^{-\frac{(t-4\tau)^2}{2\tau^2}} \ , \tag{10}$$

where $H(t)$ is the Heaviside distribution and where $\tau = 0.045$ s.

A seismic recording station is located at x = 10000 m and y = 10000 m at the surface of the model (z = 0 m). It records the three components of the velocity vector, with the X axis of the grid oriented towards the East, Y towards the North, and Z up. CPML absorbing layers with a thickness of 10 grid points are implemented on all the edges of the grid except the free surface. In some cases below we will also run a simulation with a thickness of 16 grid points for the CPML and show that on a GPU it is actually faster than when using 10 grid points.

Let us first validate the GPU single-precision implementation of our finite-difference code by comparing the results to an existing and already validated (Aochi & Douglas 2006) serial single-precision implementation in C for a classical processor core. In Figure 5 we compare the time evolution of the three components of the velocity vector recorded at the seismic station. The two sets of seismograms for each component are visually almost identical, and the absolute difference amplified by a factor of 5,000 is small, which validates the implementation on the GPU. Very small differences are observed owing to the fact that the GPU and CPU implementations perform the operations in a different order, which results in a different roundoff. Also, single precision arithmetic on current GPU hardware is not fully compliant with the IEEE-754 s23e8 standard because of slightly higher errors (ulp–units in last place), the lack of denormalization and rounding that always occurs to zero. This may result in additional very small differences between operations performed on a GPU and on a CPU. Let us note that it will not be the case any more of the FERMI hardware of NVIDIA, which is fully IEEE-754 compliant (NVIDIA Corporation 2009b).

Let us now study the performance of the GPU code running on a single GPU and compare it to the performance of the CPU code running on a single CPU. Let us start with a 8800 GTX card. We determined experimentally that out of the 768 megabytes of memory that are installed on that GPU, approximately 708 megabytes are available to the user. The largest grid that we can thus use has a size of 384 $\times$ 328 $\times$ 52 points because when run with CPMLs with a thickness of 16 points it requires 700 megabytes of memory on the GPU to store the model and all the finite-difference arrays; with CPMLs with a thickness of 10 points this reduces to 655 megabytes.

We compile the reference serial C code with the GNU gcc compiler version 3.4.6 with option -O3, and the code for the GPU with the standard NVIDIA nvcc compiler of CUDA version 2.2. We also compiled the code with the Portland pgcc compiler, but in the case of our single-precision application GNU gcc resulted in faster code and we thus selected it. The total elapsed time to perform the simulation (being the only user on the machine, to avoid interferences with other jobs running and belonging to other users) is 6656 s for the CPU code running on one CPU core and 177 s for the GPU code running on one GPU.

It is uneasy to define the acceleration factor, also called ‘speedup’, that one can get by resorting to using GPUs. There is currently a debate in the computer science community about how this can or should be done, and there is no unique definition. The easiest possibility is to define it as the ratio between the time spent running the whole simulation on a single CPU core and the time spent running the same simulation on a single GPU. But one can then argue that modern compute nodes have several CPUs, and that each of these CPUs contains several CPU cores, which are not used in that definition. Therefore, another possible definition of the speedup is the ratio between the time spent running the whole simulation on all the CPU cores of a compute node and the time spent running the same simulation on all the GPUs installed on that compute node; this requires having two parallel codes, one that can run on multiple CPUs and another one that can run on multiple GPUs. A third definition uses a mixed (often called ‘hybrid’) model by taking the ratio between the time spent running the whole simulation on all the CPU cores of a compute node and the time spent running the same simulation on all the GPUs plus all the CPU cores (simultaneously) installed on that compute node; i.e., using all the resources available on a compute node in both cases, with ot without the GPUs. But that requires designing a parallel GPU code that can use multiple GPUs but also do part of the calculations on the multiple CPUs and CPU cores on which these GPUs are installed (hence the name ‘hybrid calculation’). Designing such a hybrid code is complex. Each of these possible definitions has advantages and drawbacks; some are more favorable to the GPU technology while others are more favorable to the CPU technology. Here for simplicity we use the first definition: the ratio between the time spent on a single CPU core and the time spent on a single GPU. The difficulty to define the acceleration factor comes from the fact that one compares two very different hardware technologies (CPU and GPU) and programming philosophies, which in addition can be mixed in hybrid models. Furthermore, compilers for CPUs and for GPUs are also very different. Moreover, for a given CPU different compilers can lead to significantly different reference execution times. In addition, giving speedup values alone is probably not sufficient because, when comparing two different technologies, one should also compare the manpower involved in code development (which is currently generally higher for GPUs) as well as electrical energy

consumption (which is also currently higher for PCs equipped with graphics accelerators). As a result, in general speedup values should be considered with caution.

With the first definition above, the speedup obtained for our application is 6656 / 177 = 38×. In Table 2 we also show performance results for thicker, i.e., more accurate CPML layers having a thickness of 16 grid points because multiples of 16 give significantly more efficient memory accesses on the GPU as seen above. Indeed we observe that the GPU code with thicker CPMLs is faster and has a higher speedup (51×) than when 10 grid points are used. For comparison we also show the case with no CPML absorbing layers, for which we get a speedup of 38×. The speedup is very high in the three cases, but it depends on the thickness of the CPML and on the size of the grid. Because we decompose the domain into slices of size 16×8 in the two horizontal directions, each block of threads is in charge of computing one of these slices. Thus, if the thickness of the CPMLs is exactly 16, all the half-warps of the thread blocks that are in charge of computing the CPMLs are non divergent, i.e., all the threads of that half-warp perform the same number of calculations. We therefore get maximum performance when the thickness of the CPMLs is a multiple of 16 and when the number of points of the grid is a multiple of 16 along X and of 8 along Y.

Let us now analyze the so-called 'strong scaling' of the GPU code, defined as the variation of the total time needed to run the application on the GPU when the number of grid points, i.e., the amount of work to perform, is increased linearly. Ideally the strong scaling curve should thus be linear. In order to avoid measuring the influence of divergent warps, i.e., in order to measure scaling only, we perform all the measurements with CPML layers having a thickness of 16 grid points, i.e., the best case of Table 2. In Figure 6, we plot the execution time of the code on the GPU as a function of the number of points along Z. We observe that when the number of grid points along the Z axis of the grid increases the execution time varies almost linearly, i.e., scaling is almost perfect.

In Figure 7 we represent the variation of the time spent by each block of threads to compute its share of the calculations on the GPU when the number of thread blocks varies along the X and Y axes of the grid. To do so, we make the total size of the model vary along the X and Y axes but keep a constant size for each block of threads, and we also use only model sizes that are multiples of the size of the basic thread blocks in order to keep all threads active. In Figure 7 we therefore express the total size of each grid axis equivalently as a number of thread blocks. Since the size of each thread block and thus the number of calculations that it performs does not vary, in an ideal case we should get a flat surface. But in practice performance per block is higher when we use more blocks, i.e., a larger finite-difference grid, because the scheduler of the GPU graphics card then has more opportunities to overlap the latency of accesses of blocks to global memory by calculations performed by other blocks that are ready to start computing (i.e., that are already done accessing global memory). However, despite the coalesced memory accesses that we implemented, scaling along X is not regular and suffers from moderate variations that are likely due to undocumented hardware requirements.

To show that speedup values can greatly vary and are sensitive to many factors, let us perform a few similar measurements on a GT200 card. We use the grid composed of $540 \times 484 \times 189$ points that requires 3.9 gigabytes of GPU memory. We run two sets of simulations: a first set with CPML layers that have a thickness of 10 grid points (thus 12.57% of the grid points are located in CPML layers) and a second set with no CPMLs. To study the sensitivity of speedup measurements to the compilers used, we run the CPU code twice: first compiled with the GNU gcc compiler (with options -O3 -fno-trapping-math) and second compiled with the Intel icc compiler (with options -O3 -ftz). For the GT200 GPU we use the NVIDIA nvcc compiler (with no particular option, because optimal options are selected automatically). As above, we make measurements over a total of 1000 time steps. In the case of CPML layers with a thickness of 10 points, the total duration of the run is 42,023 seconds on a CPU core with the GNU gcc compiler, 13,326 seconds on a CPU core with the Intel icc compiler, and 770 seconds on a GPU. Thus, with the first definition above, the speedup is 42023 / 770 = 54.6x and 13326 / 770 = 17.3x, respectively. When we do not implement CPML layers, the total duration of the run is 20,941 seconds on a CPU core with the GNU gcc compiler, 10,475 seconds on a CPU core with the Intel icc compiler, and 360 seconds on a GPU. The speedup is then 20941 / 360 = 58.2x and 10475 / 360 = 29.1x, respectively.

Let us now analyze the behavior of the code when several GPUs are used simultaneously. We use 4, 6, 8, 9, 12, 16, 20, or 25 GT200 GPUs, each managed by a different CPU processor core, i.e., we also use 4, 6, 8, 9, 12, 16, 20, or 25 CPU cores. Communications between the different CPU cores are handled based upon MPI message passing, as explained in Section 3.3.

In Figure 8 we first perform a so-called 'weak scaling' test, i.e., a performance scaling test in which the amount of work to perform per GPU is kept approximately constant. Each CPU core + GPU thus handles a 4 GB mesh slice and we increase the number of GPUs that participate in the calculation, which implies that we increase the total size of the mesh. Such a weak scaling test allows one to see if communications between nodes are effectively overlapped with calculations on the GPUs, in which case ideally the weak scaling curve should be flat (i.e., the performance level should remain constant when more nodes participate in the calculation because the amount of work to perform on each node remains approximately constant). In Figure 9 we then perform a so-called 'strong scaling' test, i.e., the total size of the mesh is now kept constant when more nodes participate in the calculation, which implies that the size of the mesh handled by each GPU decreases. Let us mention that in these scaling tests we do not include CPML absorbing layers because in these layers a significantly different (higher) number of operations is performed (to solve more equations), which would result in an imbalance between nodes and thus measurements that would be very difficult to interpret.

We observe that the weak scaling is very good, whereas the strong scaling is not as good because when we use 25 GPUs for instance instead of 4 for a model size of 4 GB, each node computes a model of size 160 MB only, which is far too small for the GPU to use all its resources efficiently enough; and let us recall that in Figure 7 we have also shown that performance is sensitive to the size of the model along X and Y. In these tests, we measured a maximum of 1.5% of elapsed time spent waiting for communications, i.e., wasted in the

`MPI_Wait()` function of Figure 4. This time fluctuates between 0% and 1.5%, even for the small 160 MB case, which leads us to say that communications seem to be efficiently overlapped, and almost negligible in practice. All the operations that have been added in Figure 4 to manage the creation, transfer and update of the MPI communication buffers take a total of about 0.5% of elapsed time, which is also negligible in practice.

5 CONCLUSIONS AND FUTURE WORK

We have accelerated a three-dimensional finite-difference wave propagation code by a factor between about 20 and 60 compared to a serial implementation using either one or several NVIDIA GPU graphics cards and the CUDA programming language. We have simulated seismic wave propagation in the heterogeneous elastic case, using CPML absorbing layers on the fictitious edges of the grid and implementing the finite-difference parallelization technique for GPUs of Micikevicius (2009), with the additional use of texture fetching in CUDA to compensate for the lack of shared memory on the graphics card, and with the use of message passing (MPI) when several GPUs are used in parallel.

We have shown that the GPU code is accurate by comparing our results to results obtained by running the same simulation on a CPU core.

Let us mention that due to the fact that the seismic wave equation written in velocity vector and stress tensor has the same second-order linear hyperbolic form as Maxwell's equations written in $\mathbf{E}$ and $\mathbf{B}$ (or $\mathbf{H}$) vectors, the GPU implementation presented can be applied to Maxwell's equations as well in a straightforward fashion.

We have mentioned that more calculations are performed in the CPML layers because more equations with more terms need to be computed there, and therefore on a parallel machine with a multi-GPU setup and MPI communications used between nodes of the cluster significant load balancing issues may arise. They may slow down the whole code, which synchronizes on the slowest computing element. In future work a possible solution would be to resort to domain decomposition with different weights assigned to CPML grid points and to regular grid points.

In future work we would also like to use the OpenCL programming language (see for instance Kirk & Hwu (2010)) instead of CUDA to make the code portable to non-NVIDIA hardware, including multi-core systems. Other options to investigate could be the use of compiler directives, somewhat similar to the philosophy of OpenMP, or higher-lever programming environments. Examples are the HMPP directives (Dolbeau et al. 2007), the StarSs project (Planas et al. 2009), StarPU (Augonnet et al. 2009) and S_GPU (Genovese et al. 2009). Another interesting issue to analyze would be how the new GPU architecture from NVIDIA, which is called the 'Fermi' (NVIDIA Corporation 2009b), may help to improve performance, in particular because this chip almost quadruples the amount of shared memory per multiprocessor and has a coherent memory cache.

Regarding the seismic wave equation itself, attenuation (viscoelasticity) should be added, which is easy to do in finite-difference codes in the time domain based upon so-called memory variables (see e.g. Kristek & Moczo (2003); Moczo & Kristek (2005); Martin & Komatitsch (2009)).

ACKNOWLEDGMENTS

The authors thank Dominik Göddeke, Gordon Erlebacher, Rached Abdelkhalek, Henri Calandra, Jean Roman, Jean-François Méhaut, Christophe Merlet, Philippe Thierry and Roland Martin for fruitful discussions about GPU computing. They acknowledge the main developers of the ONDES3D software package, Hideo Aochi, Ariane Ducellier and Yohan Lee-Tin-Yien from BRGM (France), for their support. They also thank Peter Moczo, an anonymous reviewer and editor Jeannot Trampert for fruitful comments that improved the manuscript. Half of the calculations were performed on a 8800 GTX system at BRGM and half on the 'Titane' BULL Novascale R422 GPU cluster at CCRT/CEA/GENCI in Bruyères-le-Châtel, France, with support from Stéphane Requena, Christine Ménaché, Édouard Audit, Jean-Noël Richet, Gilles Wiber, Julien Derouillat, Laurent Nguyen and Pierre Bonneau.

REFERENCES

Abdelkhalek, R., 2007. *Évaluation des accélérateurs de calcul GPGPU pour la modélisation sismique*, Master's thesis, ENSEIRB, Bordeaux, France.

Abdelkhalek, R., Calandra, H., Coulaud, O., Roman, J., & Latu, G., 2009. Fast seismic modeling and reverse time migration on a GPU cluster, in *High Performance Computing & Simulation 2009*, pp. 36–44, Leipzig, Germany, http://hal.inria.fr/docs/00/40/39/33/PDF/hpcs.pdf.

Adams, S., Payne, J., & Boppana, R., 2007. Finite difference time domain (FDTD) simulations using graphics processors, in *Proceedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference*, pp. 334–338, IEEE Computer Society, Washington, DC, USA, Pittsburgh, Pennsylvania, USA.

Alterman, Z. & Karal, F. C., 1968. Propagation of elastic waves in layered media by finite difference methods, *Bull. Seismol. Soc. Am.*, **58**, 367–398.

Anderson, J. A., Lorenz, C. D., & Travesset, A., 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units, *J. Comput. Phys.*, **227**(10), 5342–5359.

Aochi, H. & Douglas, J., 2006. Testing the validity of simulated strong ground motion from the dynamic rupture of a finite fault by using empirical equations, *Bull. Earthq. Eng.*, **4**(3), 211–229.

Augonnet, C., Thibault, S., Namyst, R., & Wacrenier, P.-A., 2009. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, in *Proceedings of the 15th EuroPar Conference*, vol. 5704 of **Lecture Notes in Computer Science**, pp. 863–874, Springer, Delft, The Netherlands.

Balevic, A., Rockstroh, L., Li, W., Hillebrand, J., Simon, S., Tausendfreund, A., Patzelt, S., & Goch, G., 2008a. Acceleration of a finite-difference method with general purpose GPUs: Lesson learned, in *Proceedings of the 8th IEEE International Conference on Computer and Information Technology*, pp. 291–294, IEEE Computer Society, Washington, DC, USA, Sydney, Australia.

Balevic, A., Rockstroh, L., Tausendfreund, A., Patzelt, S., Goch, G., & Simon, S., 2008b. Accelerating simulations of light scattering based on a finite-difference time-domain method with general purpose GPUs, in *Proceedings of the 11th IEEE International Conference on Computational Science and Engineering*, pp. 327–334, IEEE Computer Society, Washington, DC, USA, Sao Paulo, Brazil.

Baron, G. S., Sarris, C. D., Fiume, E., & Rogers Sr., E. S., 2005. Fast and accurate time-domain simulations with commodity graphics hardware, in *Proceedings of the 2005 IEEE Antennas and Propagation Society International Symposium*, vol. 4A, pp. 193–196, IEEE Computer Society, Washington, DC, USA, Seoul, Korea.

Bell, N. & Garland, M., 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors, in *SC'09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, pp. 1–11, ACM, New York, USA.

Bérenger, J. P., 1994. A Perfectly Matched Layer for the absorption of electromagnetic waves, *J. Comput. Phys.*, **114**, 185–200.

Bernacki, M., Lanteri, S., & Piperno, S., 2006. Time-domain parallel simulation of heterogeneous wave propagation on unstructured grids using explicit, nondiffusive, discontinuous Galerkin methods, *J. Comput. Acoust.*, **14**(1), 57–81.

Brandvik, T. & Pullan, G., 2007. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware, *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, **221**(12), 1745–1748.

Carcione, J. M. & Wang, P. J., 1993. A Chebyshev collocation method for the wave equation in generalized coordinates, *Comp. Fluid Dyn. J.*, **2**, 269–290.

Chaljub, E., Komatitsch, D., Vilotte, J. P., Capdeville, Y., Valette, B., & Festa, G., 2007. Spectral element analysis in seismology, in *Advances in wave propagation in heterogeneous media*, vol. 48 of **Advances in Geophysics**, pp. 365–419, eds Wu, R.-S. & Maupin, V., Elsevier - Academic Press, London, UK.

Chew, W. C. & Liu, Q., 1996. Perfectly Matched Layers for elastodynamics: a new absorbing boundary condition, *J. Comput. Acoust.*, **4**(4), 341–359.

Chorin, A. J., 1968. Numerical solution of the Navier-Stokes equations, *Math. Comput.*, **22**, 745–762.

Cockburn, B., Karniadakis, G. E., & Shu, C.-W., 2000. *Discontinuous Galerkin Methods: Theory, Computation and Applications*, Springer, Heidelberg, Germany.

Collino, F. & Tsogka, C., 2001. Application of the PML absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media, *Geophysics*, **66**(1), 294–307.

Danielson, K. T. & Namburu, R. R., 1998. Nonlinear dynamic finite element analysis on parallel computers using Fortran90 and MPI, *Advances in Engineering Software*, **29**(3-6), 179–186.

Dolbeau, R., Bihan, S., & Bodin, F., 2007. HMPP: A hybrid multi-core parallel programming environment, in *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'2007)*, pp. 1–5, Boston, Massachusetts, USA.

Drossaert, F. H. & Giannopoulos, A., 2007. A nonsplit complex frequency-shifted PML based on recursive integration for FDTD modeling of elastic waves, *Geophysics*, **72**(2), T9–T17.

Dumbser, M. & Käser, M., 2006. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes-II. The three-dimensional isotropic case, *Geophys. J. Int.*, **167**(1), 319–336.

Elsen, E., LeGresley, P., & Darve, E., 2008. Large calculation of the flow over a hypersonic vehicle using a GPU, *J. Comput. Phys.*, **227**(24), 10148–10161.

Falk, R. S. & Richter, G. R., 1999. Explicit finite element methods for symmetric hyperbolic equations, *SIAM Journal on Numerical Analysis*, **36**(3), 935–952.

Festa, G. & Vilotte, J. P., 2005. The Newmark scheme as velocity-stress time-staggering: an efficient PML implementation for spectral-element simulations of elastodynamics, *Geophys. J. Int.*, **161**, 789–812.

Festa, G., Delavaud, E., & Vilotte, J. P., 2005. Interaction between surface waves and absorbing boundaries for wave propagation in geological basins: 2D numerical simulations, *Geophys. Res. Lett.*, **32**(20), L20306.

Gedney, S. D. & Zhao, B., 2010. An auxiliary differential equation formulation for the complex-frequency shifted PML, *IEEE Transactions on Antennas and Propagation*, **58**(3), 838–847.

Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.-F., Neelov, A., & Goedecker, S., 2009. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures, *Journal of Chemical Physics*, **131**(3), 034103.

Giraldo, F. X., Hesthaven, J. S., & Warburton, T., 2002. Nodal high-order discontinuous Galerkin methods for the spherical shallow water equations, *J. Comput. Phys.*, **181**(2), 499–525.

Graves, R. W., 1996. Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences, *Bull. Seismol. Soc. Am.*, **86**(4), 1091–1106.

Gropp, W., Lusk, E., & Skjellum, A., 1994. *Using MPI, portable parallel programming with the Message-Passing Interface*, MIT Press, Cambridge, USA.

Grote, M. J., Schneebeli, A., & Schötzau, D., 2006. Discontinuous Galerkin finite element method for the wave equation, *SIAM Journal on Numerical Analysis*, **44**(6), 2408–2431.

Hastings, F. D., Schneider, J. B., & Broschat, S. L., 1996. Application of the Perfectly Matched Layer (PML) absorbing boundary condition to elastic wave propagation, *J. Acoust. Soc. Am.*, **100**(5), 3061–3069.

Holberg, O., 1987. Computational aspects of the choice of operator and sampling interval for numerical differentiation in large-scale simulation of wave phenomena, *Geophys. Prospect.*, **35**, 629–655.

Hu, F. Q., Hussaini, M. Y., & Rasetarinera, P., 1999. An analysis of the discontinuous Galerkin method for wave propagation problems, *J. Comput. Phys.*, **151**(2), 921–946.

Humphrey, J. R., Price, D. K., Durbano, J. P., Kelmelis, E. J., & Martin, R. D., 2006. High performance 2D and 3D FDTD solvers on GPUs, in *Proceedings of the 10th WSEAS International Conference on Applied Mathematics*, pp. 547–550, World Scientific and Engineering Academy and Society (WSEAS), Dallas, Texas, USA.

Inman, M. J. & Elsherbeni, A. Z., 2008. Optimization and parameter exploration using GPU based FDTD solvers, in *Proceedings of the 2008 IEEE MTT-S International Microwave Symposium*, pp. 149–152, Atlanta, Georgia, USA.

Inman, M. J., Elsherbeni, A. Z., Maloney, J. G., & Baker, B. N., 2007. GPU based FDTD solver with CPML boundaries, in *Proceedings of the 2007 IEEE Antennas and Propagation Society International Symposium*, pp. 5255–5258, Honolulu, Hawaii, USA.

Kawase, H., 1988. Time-domain response of a semi-circular canyon for incident *SV*, *P* and Rayleigh waves calculated by the discrete wavenumber boundary

element method, *Bull. Seismol. Soc. Am.*, **78**, 1415–1437.

Kirk, D. B. & Hwu, W.-m. W., 2010. *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, Boston, Massachusetts, USA.

Klöckner, A., Warburton, T., Bridge, J., & Hesthaven, J. S., 2009. Nodal discontinuous Galerkin methods on graphics processors, *J. Comput. Phys.*, **228**, 7863–7882.

Komatitsch, D. & Martin, R., 2007. An unsplit convolutional Perfectly Matched Layer improved at grazing incidence for the seismic wave equation, *Geophysics*, **72**(5), SM155–SM167.

Komatitsch, D. & Tromp, J., 1999. Introduction to the spectral-element method for 3-D seismic wave propagation, *Geophys. J. Int.*, **139**(3), 806–822.

Komatitsch, D., Coutel, F., & Mora, P., 1996. Tensorial formulation of the wave equation for modelling curved interfaces, *Geophys. J. Int.*, **127**(1), 156–168.

Komatitsch, D., Michéa, D., & Erlebacher, G., 2009. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA, *Journal of Parallel and Distributed Computing*, **69**(5), 451–460.

Komatitsch, D., Erlebacher, G., Göddeke, D., & Michéa, D., 2010a. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster, *J. Comput. Phys.*, **229**(20), 7692–7714.

Komatitsch, D., Göddeke, D., Erlebacher, G., & Michéa, D., 2010b. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs, *Computer Science Research and Development*, **25**(1-2), 75–82.

Krakiwsky, S. E., Turner, L. E., & Okoniewski, M. M., 2004a. Graphics processor unit (GPU) acceleration of a finite-difference time-domain (FDTD) algorithm, *Proceedings of the 2004 IEEE International Symposium on Circuits and Systems*, pp. 265–268.

Krakiwsky, S. E., Turner, L. E., & Okoniewski, M. M., 2004b. Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU), *IEEE 2004 MTT-S International Microwave Symposium Digest*, **2**, 1033–1036.

Kristek, J. & Moczo, P., 2003. Seismic-wave propagation in viscoelastic media with material discontinuities: A 3D fourth-order staggered-grid finite-difference modeling, *Bull. Seismol. Soc. Am.*, **93**(5), 2273–2280.

Kristek, J., Moczo, P., & Galis, M., 2009. A brief summary of some PML formulations and discretizations for the velocity-stress equation of seismic motion, *Studia Geophysica et Geodaetica*, **53**(4), 459–474.

Liu, Q., Polet, J., Komatitsch, D., & Tromp, J., 2004. Spectral-element moment tensor inversions for earthquakes in Southern California, *Bull. Seismol. Soc. Am.*, **94**(5), 1748–1761.

Madariaga, R., 1976. Dynamics of an expanding circular fault, *Bull. Seismol. Soc. Am.*, **66**(3), 639–666.

Martin, R. & Komatitsch, D., 2006. An optimized convolution-perfectly matched layer (C-PML) absorbing technique for 3D seismic wave simulation based on a finite-difference method, *Geophysical Research Abstracts*, **8**, 03988, Abstract EGU06-A-03988, www.cosis.net/abstracts/EGU06/03988/EGU06-J-03988-1.pdf.

Martin, R. & Komatitsch, D., 2009. An unsplit convolutional perfectly matched layer technique improved at grazing incidence for the viscoelastic wave equation, *Geophys. J. Int.*, **179**(1), 333–344.

Martin, R., Komatitsch, D., & Barucq, H., 2005. An optimized convolution-perfectly matched layer (C-PML) absorbing technique for 3D seismic wave simulation based on a finite-difference method, *Eos Trans. AGU*, **86**(52), Fall Meet. Suppl., Abstract NG43B–0574, www.agu.org/meetings/fm05/waisfm05.html.

Martin, R., Komatitsch, D., & Ezziani, A., 2008a. An unsplit convolutional perfectly matched layer improved at grazing incidence for seismic wave equation in poroelastic media, *Geophysics*, **73**(4), T51–T61.

Martin, R., Komatitsch, D., & Gedney, S. D., 2008b. A variational formulation of a stabilized unsplit convolutional perfectly matched layer for the isotropic or anisotropic seismic wave equation, *Comput. Model. Eng. Sci.*, **37**(3), 274–304.

Martin, R., Komatitsch, D., Gedney, S. D., & Bruthiaux, E., 2010. A high-order time and space formulation of the unsplit perfectly matched layer for the seismic wave equation using Auxiliary Differential Equations (ADE-PML), *Comput. Model. Eng. Sci.*, **56**(1), 17–42.

Meza-Fajardo, K. C. & Papageorgiou, A. S., 2008. A nonconvolutional, split-field, perfectly matched layer for wave propagation in isotropic and anisotropic elastic media: Stability analysis, *Bull. Seismol. Soc. Am.*, **98**(4), 1811–1836.

Micikevicius, P., 2009. 3D finite-difference computation on GPUs using CUDA, in *GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, Washington, DC, USA.

Moczo, P. & Kristek, J., 2005. On the rheological models used for time-domain methods of seismic wave propagation, *Geophys. Res. Lett.*, **32**, L01306.

Moczo, P., Robertsson, J., & Eisner, L., 2007. The finite-difference time-domain method for modeling of seismic wave propagation, in *Advances in wave propagation in heterogeneous media*, vol. 48 of **Advances in Geophysics**, chap. 8, pp. 421–516, eds Wu, R.-S. & Maupin, V., Elsevier - Academic Press, London, UK.

Monk, P. & Richter, G. R., 2005. A discontinuous Galerkin method for linear symmetric hyperbolic systems in inhomogeneous media, *Journal of Scientific Computing*, **22-23**(1-3), 443–477.

NVIDIA Corporation, 2009a. *NVIDIA CUDA Programming Guide version 2.3*, Santa Clara, California, USA, 139 pages.

NVIDIA Corporation, 2009b. NVIDIA's next generation CUDA compute architecture: FERMI, Tech. rep., NVIDIA, Santa Clara, California, USA, 22 pages.

Nyland, L., Harris, M., & Prins, J., 2007. Fast N-body simulation with CUDA, in *GPU Gems 3*, chap. 31, pp. 677–695, Addison-Wesley Professional, Boston, MA, USA.

Owens, J. D., Luebke, D. P., Govindaraju, N. K., Harris, M. J., Krüger, J., Lefohn, A. E., & Purcell, T. J., 2007. A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum*, **26**(1), 80–113.

Planas, J., Badia, R. M., Ayguadé, E., & Labarta, J., 2009. Hierarchical task-based programming with StarSs, *International Journal of High Performance Computing Applications*, **23**(3), 284–299.

Price, D. K., Humphrey, J. R., & Kelmelis, E. J., 2007. GPU-based accelerated 2D and 3D FDTD solvers, in *Proceedings of the SPIE Physics and Simulation of Optoelectronic Devices XV Conference*, vol. 6468, SPIE and the International Society for Optical Engineering, San Jose, California, USA.

Reed, W. H. & Hill, T. R., 1973. Triangular mesh methods for the neutron transport equation, Tech. Rep. LA-UR-73-479, Los Alamos Scientific Laboratory, Los Alamos, USA.

Rivière, B. & Wheeler, M. F., 2003. Discontinuous finite element methods for acoustic and elastic wave problems, *Contemporary Mathematics*, **329**, 271–282.

Roden, J. A. & Gedney, S. D., 2000. Convolution PML (CPML): An efficient FDTD implementation of the CFS-PML for arbitrary media, *Microwave and Optical Technology Letters*, **27**(5), 334–339.

Tromp, J., Komatitsch, D., & Liu, Q., 2008. Spectral-element and adjoint methods in seismology, *Communications in Computational Physics*, **3**(1), 1–32.

Vai, R., Castillo-Covarrubias, J. M., Sánchez-Sesma, F. J., Komatitsch, D., & Vilotte, J. P., 1999. Elastic wave propagation in an irregularly layered medium, *Soil Dynamics and Earthquake Engineering*, **18**(1), 11–18.

Valcarce, A., De La Roche, G., & Zhang, J., 2008. A GPU approach to FDTD for radio coverage prediction, in *Proceedings of the 11th IEEE International*

Term	Explanation
Host	The Central Processing Unit (CPU), i.e., a classical processor (in our case, one core of a multicore processor).
Device	The Graphics Processing Unit (GPU), i.e., the graphics card.
Kernel	A function executed in parallel on the device.
Thread block	A set of threads with common access to a shared memory area– all the threads within a block can be synchronized.
Grid	A set of thread blocks - a kernel is executed on a grid of thread blocks.
Warp	A group of 32 threads executed concurrently on a multiprocessor of the GPU.
Multiprocessor occupancy	The ratio of the actual number of active warps on a multiprocessor to the maximum number of active warps allowed.
Global memory	Uncached off-chip DRAM memory.
Shared memory	High-performance on-chip register memory, limited to 16 kB on the hardware we use.
Constant memory	A read-only region of device memory with faster access times and a cache mechanism.
Coalesced memory accesses	Simultaneous GPU global memory accesses coalesced into a single contiguous, aligned memory access at the scope of a half-warp.

Table 1. Glossary of some terms used in CUDA programming and in this article. For more details the reader is referred to the CUDA documentation NVIDIA Corporation (2009a) and GPU/CUDA conference tutorials (see e.g. http://gpgpu.org/developer).

Conference on Communication Systems, pp. 1585–1590, Guangzhou, China.

Virieux, J., 1986. *P-SV* wave propagation in heterogeneous media: velocity-stress finite-difference method, *Geophysics*, **51**, 889–901.

Yang, J., Wang, Y., & Chen, Y., 2007. GPU accelerated molecular dynamics simulation of thermal conductivities, *J. Comput. Phys.*, **221**, 799–804.

Yee, K. S., 1966. Numerical solution of initial boundary value problems involving Maxwell's equations, *IEEE Transactions on Antennas and Propagation*, **14**, 302–307.

CPML thickness	0	10	16
CPU time (s)	3840	6656	8320
GPU time (s)	100	177	162
Speedup	38	38	51

Table 2. Speedup with respect to a serial implementation obtained when running the three-dimensional finite-difference code on one GPU instead of one CPU core, i.e., ratio between the total time spent running the code on a CPU core and the total time spent running the code on a GPU. The grid is composed of 384 $\times$ 328 $\times$ 52 grid points. We perform the measurement for a grid with CPML absorbing layers that have a thickness of 10 grid points, which is a classical value used on CPUs. We also perform the measurement with thicker, i.e., more accurate CPML layers having a thickness of 16 grid points because multiples of 16 give significantly more efficient memory accesses on the GPU. Indeed we observe that the GPU code with thicker CPMLs is faster than when 10 grid points are used, which means that on a GPU it is better to use thicker absorbing layers. For comparison we also show the case with no CPML absorbing layers.

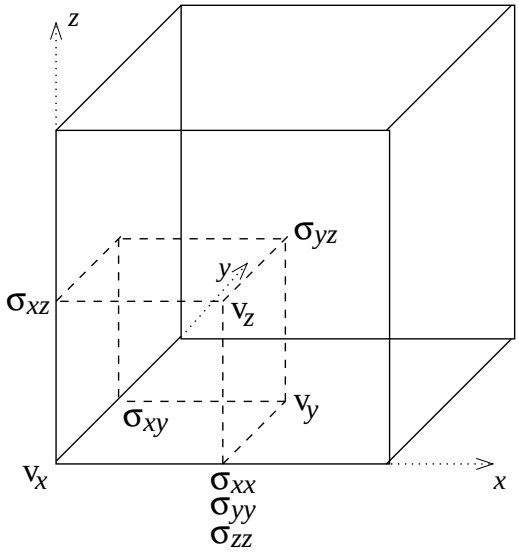

Figure 1. Elementary grid cell of the three-dimensional staggered spatial finite-difference method based on Madariaga (1976) used classically to discretize the equations of elastodynamics, using the components of the velocity vector $\mathbf{v}$ and of the symmetric stress tensor $\boldsymbol{\sigma}$ as unknowns. The unknowns are defined at grid points or half way between grid points depending on the component and of the grid axis considered. For Maxwell's equations a similar staggered grid was introduced by Yee (1966).

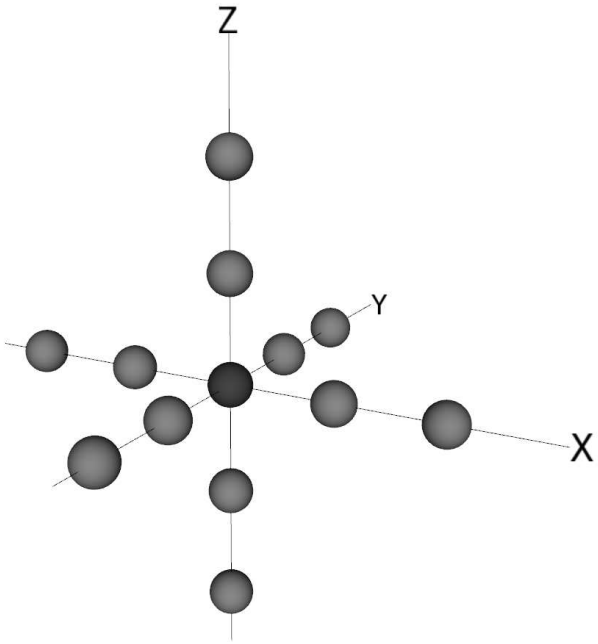

Figure 2. Illustration of the spatial stencil of the three-dimensional fourth-order finite-difference operator used to approximate spatial derivatives by a discrete difference between adjacent grid points, after discretization of the model in a grid with elementary grid cells as in Figure 1.

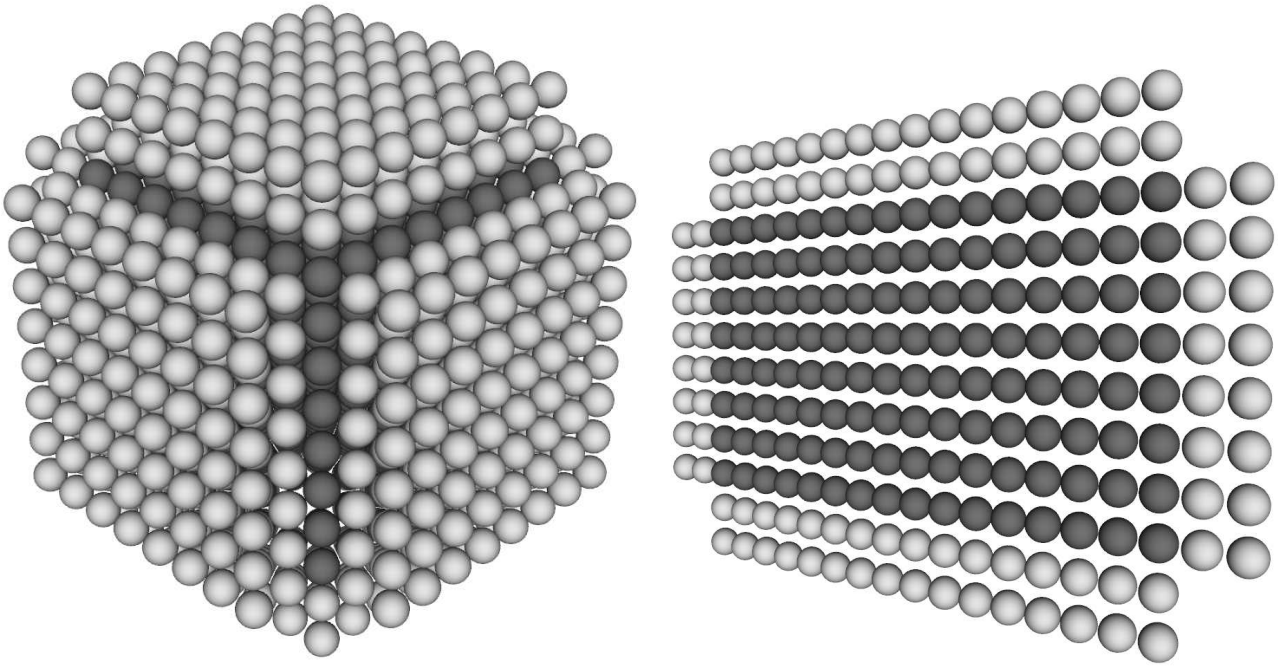

Figure 3. (Left) Three-dimensional block of $8 \times 8 \times 8 = 512$ grid points (dark gray) and its 'halo' of $6 \times 8 \times 8 \times 2 = 768$ grid points (light gray) that need to be loaded from global memory to shared memory to implement a finite-difference calculation on the GPU by a block of 512 CUDA threads based on the stencil of Figure 2 and on an intuitive decomposition into a cubic distribution of threads. (Right) Two-dimensional tile of $8 \times 16 = 128$ grid points (dark gray) and its two-dimensional 'halo' of $8 \times 2 \times 2 + 16 \times 2 \times 2 = 96$ grid points (light gray) that need to be loaded from global memory to shared memory by this block of 128 threads when the more efficient two-dimensional approach introduced by Micikevicius (2009) is used. We have not drawn the additional halo of $8 \times 16 \times 2 \times 2 = 512$ grid points located in front and behind this tile because it does not need to be stored in shared memory, it can much more efficiently be stored in registers organized in a pipeline fashion, taking advantage of the fact that access to these registers is extremely fast.

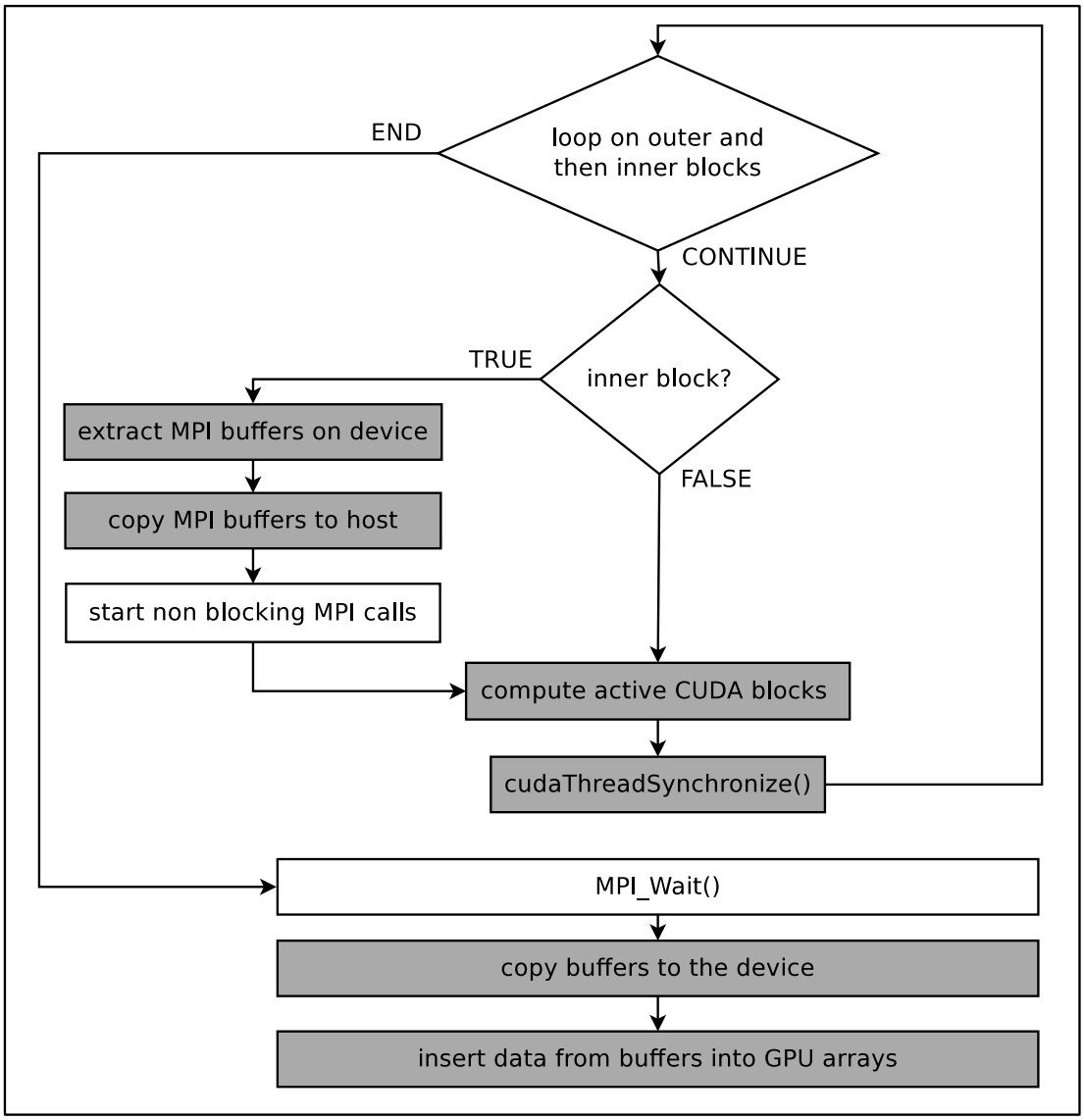

Figure 4. Implementation of asynchronous (i.e., non blocking) message-passing MPI communications to allow for overlapping of communication time between the compute nodes with calculations performed on the GPUs. The white boxes are executed on each CPU core, while the filled boxes are launched on each GPU. In the velocity-stress finite-difference algorithm there are two main computation kernels: a first to compute the stress tensor based on the components of the velocity vector, and a second to compute the velocity vector based on the components of the stress tensor. We thus use this communication pattern twice at each iteration of the time loop, once to handle stress buffers and once to handle velocity buffers.

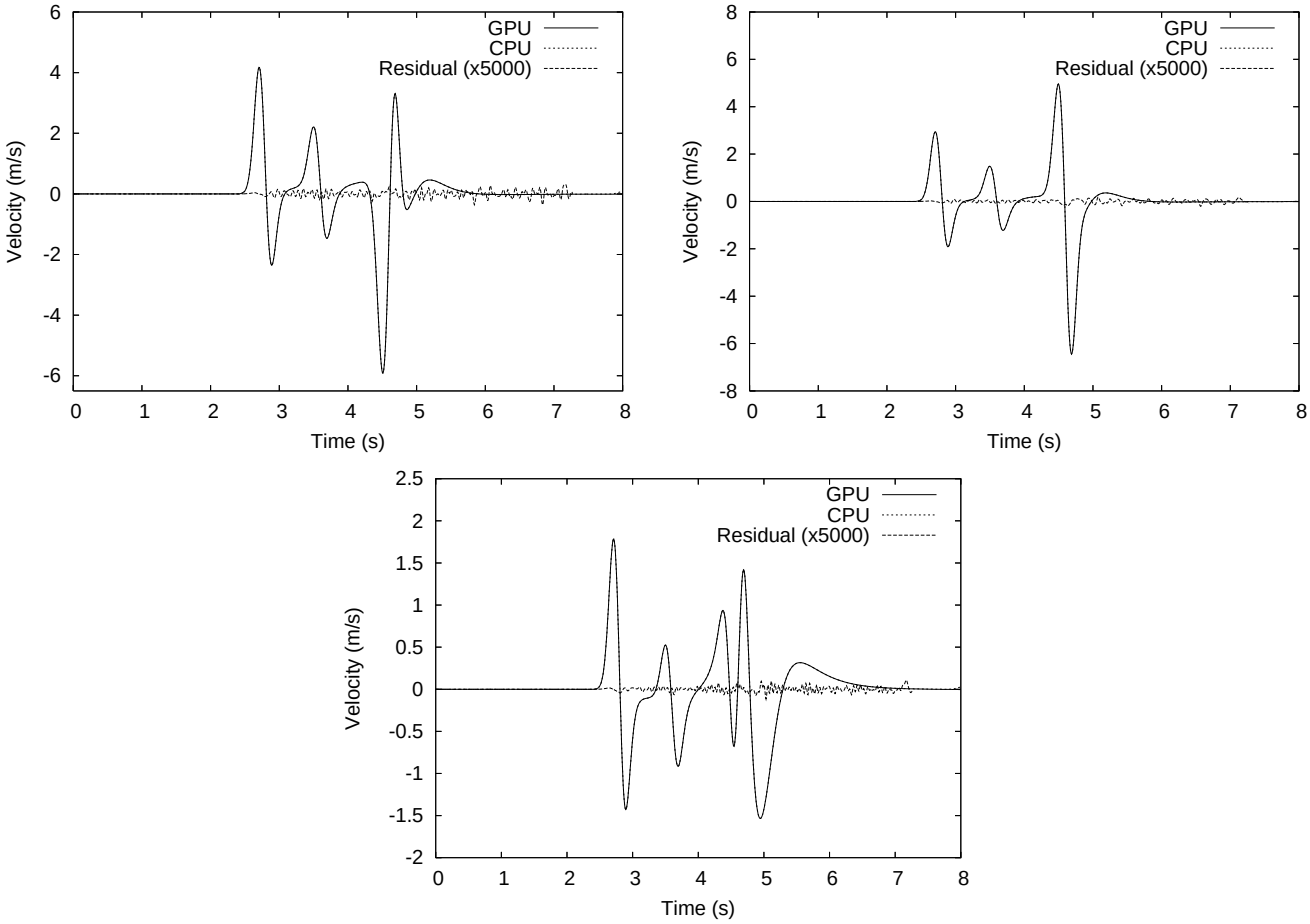

Figure 5. Time evolution of the X (top left), Y (top right) and Z (bottom) components of the velocity vector recorded at the seismic receiver with our GPU / CUDA implementation on one GPU (solid lines) and with the reference CPU / C language implementation of Aochi & Douglas (2006) on one CPU (dotted lines). The two sets of seismograms for each component are visually almost identical, which validates the implementation on the GPU. Tiny differences appear owing to the fact that the GPU and CPU implementations perform the operations in a different order, which results in a different roundoff. But the third set of curves (dashed lines) shows that the absolute difference, i.e., the so-called residual, amplified by a factor of 5,000 is small.

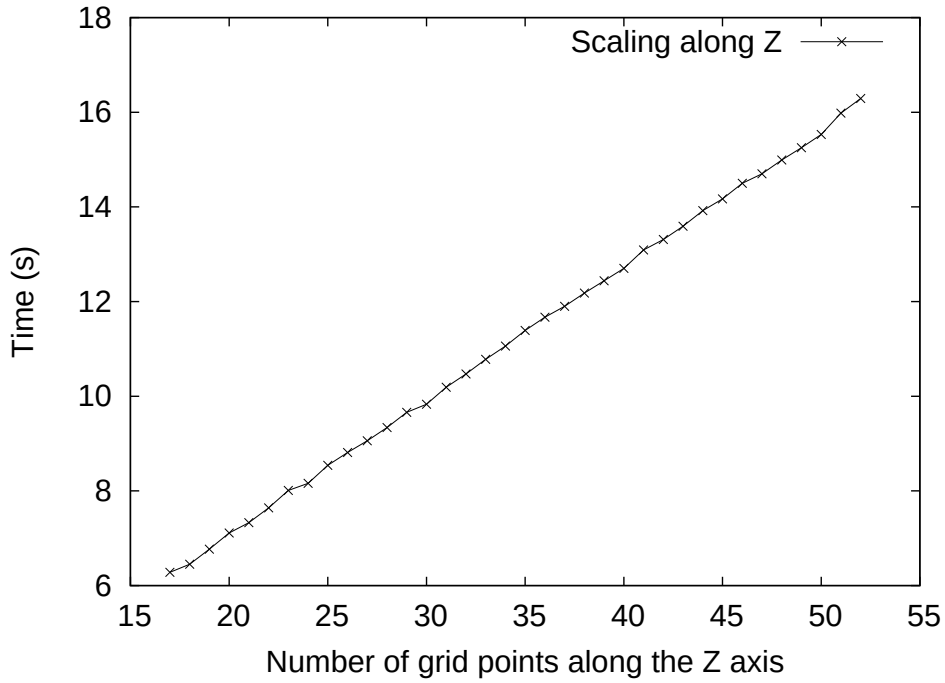

Figure 6. Strong scaling of the GPU three-dimensional finite-difference code running on a single GPU as a function of the number of points along Z, i.e., variation of the total time spent running the code on the GPU when the number of grid points along the Z axis of the grid increases. Scaling is almost linear, i.e., almost perfect.

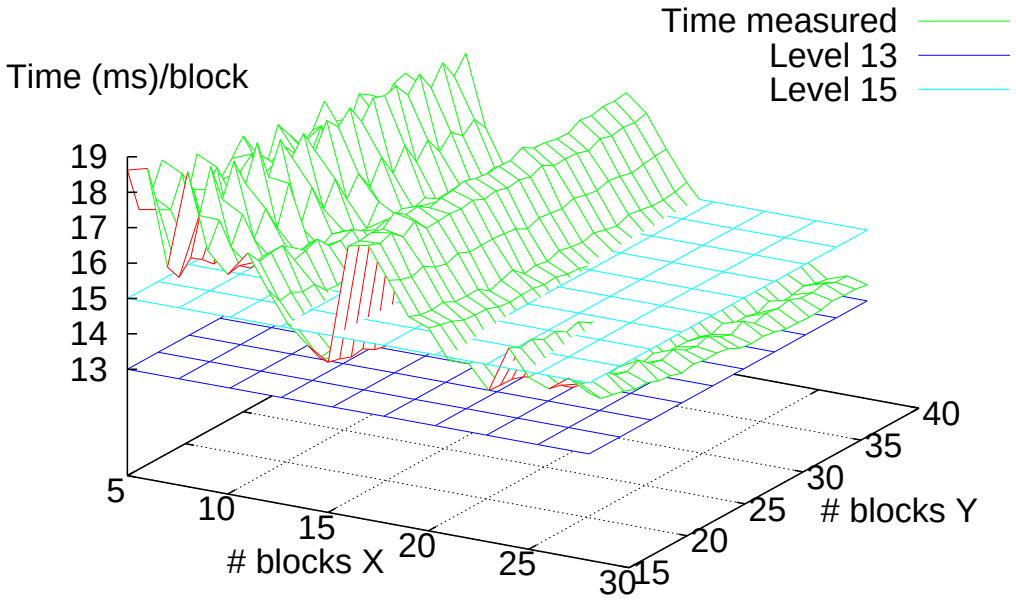

Figure 7. Scaling of the GPU three-dimensional finite-difference code running on a single GPU as a function of the number of thread blocks along the X and Y axes of the grid, i.e., variation of the time spent by each block of threads to compute its share of the calculations on the GPU when the number of thread blocks varies. The size of each thread block and thus the number of calculations that it performs does not vary, therefore in an ideal case we should get a flat surface. Here, despite the coalesced memory accesses that we implemented, scaling along X is not regular and suffers from moderate variations that are likely due to undocumented hardware requirements. The blue and cyan horizontal planes indicate reference levels for comparison.

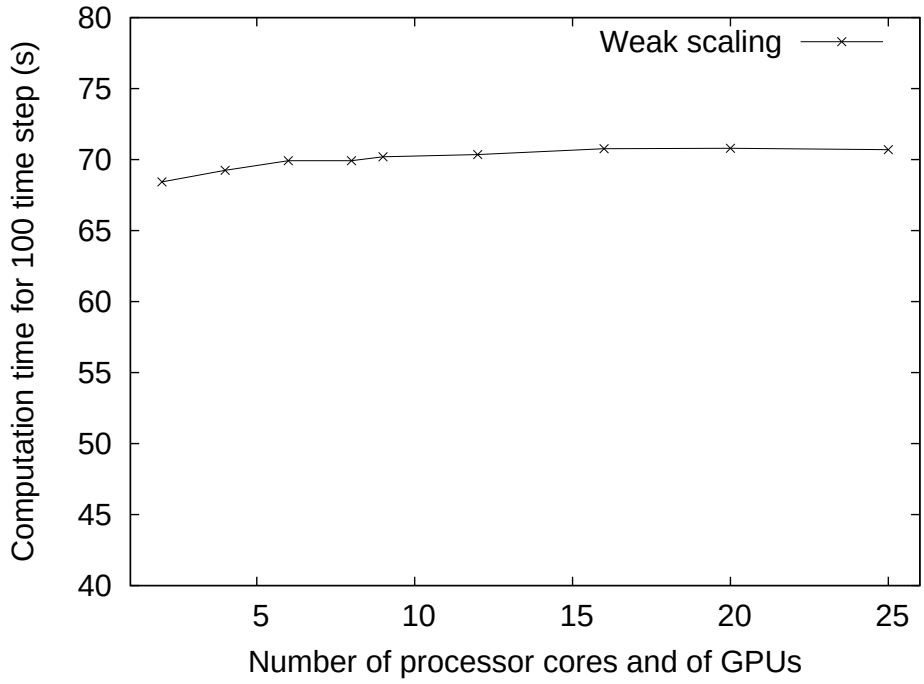

Figure 8. Weak scaling test, i.e., performance scaling test in which the amount of work to perform per GPU is kept approximately constant. Each CPU core + GPU handles a 4 GB mesh and we increase the number of GPUs that participate in the calculation. Ideally the weak scaling curve should be flat; here the measured weak scaling is very good.

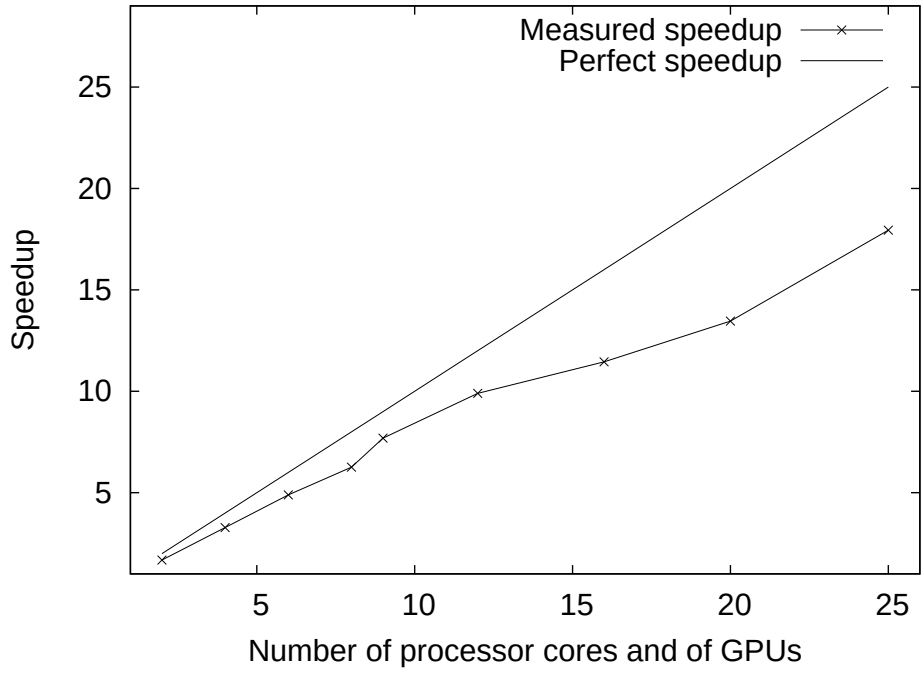

Figure 9. Strong scaling test, i.e., the total size of the mesh is kept constant when more nodes participate in the calculation. Therefore the size of the mesh slice handled by each GPU decreases. Ideally strong scaling should be linear (solid straight line), but in practice the amount of calculations performed per GPU becomes too small when a large number of GPUs is used, thus reducing measured performance (line with symbols).